Reversible Programming: A Case Study of Two String-Matching Algorithms

Robert Glück
DIKU, Department of Computer Science,
University of Copenhagen, Denmark
glueck@acm.org

Tetsuo Yokoyama
Dept. of Electronics and Communication Technology,
Nanzan University, Japan
tyokoyama@acm.org

String matching is a fundamental problem in algorithm. This study examines the development and construction of two reversible string-matching algorithms: a naive string-matching algorithm and the Rabin–Karp algorithm. The algorithms are used to introduce reversible computing concepts, beginning from basic reversible programming techniques to more advanced considerations about the injectivization of the polynomial hash-update function employed by the Rabin–Karp algorithm. The results are two clean input-preserving reversible algorithms that require no additional space and have the same asymptotic time complexity as their classic irreversible originals. This study aims to contribute to the body of reversible algorithms and to the discipline of reversible programming.

1 Introduction

Reversible computing is an unconventional computing paradigm in which all computations are forward and backward deterministic. It complements existing mainstream programming paradigms that are forward deterministic, but usually backward nondeterministic, such as imperative and functional programming languages [9]. Reversible computing is required when the deletion of information is considered harmful as in quantum-based computing and to overcome Landauer's physical limit (for a summary see [6, 11]). Additionally, reversible computing is a sweet spot for studying non-standard semantics and program inversion, which concern fundamental questions regarding program transformation [7].

The contribution of this study is threefold:

- Introduce reversible computing concepts by a program development in this unconventional paradigm.
- Explain a new, efficient reversible version of the Rabin–Karp algorithm for string matching.
- Contribute to the advancement of a reversible programming discipline.

String matching is a fundamental algorithmic problem with a wide range of practical applications. The problem is stated in a few lines (we follow established terminology [4]):

> Let $T[0..n-1]$ be a *text* of length n and $P[0..m-1]$ be a *pattern* of length m $(\leq n)$ where the elements of arrays T and P are characters drawn from a finite alphabet Σ of size d. The arrays are also called *strings* of characters. A pattern P occurs with a *valid shift* s in text T if $T[s..s+m-1] = P[0..m-1]$. The *string-matching problem* is to find all valid shifts of P in T.

Today there are various string-matching algorithms all of which are defined in conventional languages. In this study, we develop efficient reversible string-matching algorithms, namely, a naive algorithm and the more efficient Rabin–Karp algorithm [10]. Although the worst-case matching time of the latter is no better than that of the naive method, the Rabin–Karp algorithm is faster on average because it uses hash values for fast, approximate matches, and only in the case of a possible match, performs

G.W. Hamilton, T. Kahsai, M. Proietti (Eds.):
Joint International Workshops on
Horn Clauses for Verification and Synthesis (HCVS 2022)
Verification and Program Transformation (VPT 2022)
EPTCS 373, 2022, pp. 1–13, doi:10.4204/EPTCS.373.1

an exact comparison of the pattern and text at the current shift. The use of a hash-update function that computes the next hash value from the current hash value (rolling hash) makes the creation of an efficient and reversible Rabin–Karp algorithm more challenging than that of the naive algorithm.

We develop the reversible string-matching algorithms to explain reversible-computing concepts and how to solve the challenge posed by Rabin–Karp's hash-based method. Specifically, we use the standard reversible programming language, Janus, with syntactic sugar to define the algorithms. Once the algorithms are written in a reversible language, they are guaranteed to be reversible.

More details about reversible computing can be found in the literature, *e.g.*, [22]. This study contributes to the existing literature on clean reversible algorithms that do not rely on tracing (*e.g.*, [2, 8]), including a reversible FFT [21], Dijkstra's permutation encoder [22], and language processors [23, 1].

2 A Reversible Naive String Matcher

We begin with the naive matcher to demonstrate how to proceed with reversible programming. The naive matcher developed in this section will later be employed in the reversible Rabin–Karp algorithm. Advanced considerations, such as the injectivization of the hash-update function, are discussed in the next section about the Rabin–Karp algorithm.

Algorithms written in a reversible language cannot delete information, but can *reversibly update* information. No deletion of information means that programs written in a reversible imperative language cannot contain assignments that overwrite values, such as `x := y + z`, only reversibly update values, such as `x += y` (shorthand for `x := x + y`).[1] Consequently, reversible languages can only be as computationally powerful as *reversible Turing machines* (RTMs) [3], which exactly compute the *computable injective functions*. This injectiveness constraint makes reversible Turing machines strictly less powerful compared to their traditional counterparts that are universal. This is a significant limitation of reversible computing; however, all non-injective functions can be *embedded* in injective functions.

There are two approaches for implementing a function in reversible language. The first approach is to begin from an implementation written in a conventional (irreversible) language and *reversibilize* it into a program written in a reversible programming language, *e.g.*, by recording the information otherwise lost (often called *garbage*). The second and the preferred approach is to change (*injectivize*) the problem specification into an injective function, which can be directly implemented in a reversible language without functional changes.

The string-matching problem has an injective specification although it is usually considered a non-injective problem that, given text T and pattern P, computes all valid shifts:

$$\mathit{match}(T,P) \quad = \quad \mathit{valid\text{-}shifts}\,. \tag{1}$$

This function specifies that T and P are consumed (deleted) by *match* and are replaced by *valid-shifts*. Considering the problem from a reversible-computing perspective, we notice that we usually do not delete T and P, but preserve them. This means that the string-matching problem has an injective specification:

$$\mathit{match}(T,P) \quad = \quad (T,P,\mathit{valid\text{-}shifts})\,. \tag{2}$$

Because of the injective specification, a faithful reversible implementation of the string-matching problem exists. This specification is an *input-preserving injectivization* (T and P are preserved).

[1] Any expression e can be used in `x += e`. Generally, to be reversible, `x` must not occur in e (*e.g.*, `x += -x` is not reversible).

A naive string-matching algorithm compares P to T at all shifts in T from left to right. When a mismatch is found at a shift s, the matching continues at the next shift $s+1$. The worst-case matching time for this (irreversible) naive matching method is $\Theta((n-m+1)m)$.

First, we consider certain standard technical details. We consider characters as integers such that alphabet Σ of size d is set $\{0,\ldots,d-1\}$. Let T and P be integer arrays of length $n+1$ and $m+1$, respectively, with terminating values $T[n] \neq P[m]$, where $P[m] \notin \Sigma$. Thus, the end of p is always signaled by a mismatch. All valid shifts s_i found during the search are pushed on a result stack. Thus, *valid-shifts* in Eqs. (1, 2) is a stack, which consists of zero or more unique indices of T.

At first glance, the reversible naive string matcher in Fig. 1 looks like a C-like program. At the second look, we notice that the program uses no destructive assignments, such as `:=`, only the reversible updates `+=` and `-=` that add to resp. subtract from a variable the value of an expression, and the conditional at lines 9 to 15 not only has an entry test at `if`, but also an exit test at `fi`, which is the point where the control flow joins after executing one of the two branches.

Reversible languages comprise elementary steps that perform injective transformations of the computation state, that is each step performs a forward and backward deterministic transition. Because the operations are reversible on the microscopic level, the macroscopic operation of a program written in a reversible language is perfectly reversible. The composition of injective functions is also an injective function, thus reversible programs implement computable injective functions. This principle is the same for all reversible languages including the transition function of a reversible Turing machine [3], a time-symmetric machine [15], and extensions that operate on quantum data (for quantum circuits, *e.g.* [16]).

A Reversible Matcher The reversible naive string matcher, shown in Fig. 1 consists of three procedures. The main procedure `naivesearch` is called with a text `T`, a pattern `P`, and an initially empty stack `R` as input. When it returns, all valid shifts are stored in `R`, and `T` and `P` are unchanged (all three arguments are pass-by-reference). The procedure tries all the possible shifts from left to right by incrementing `s` from `0` to `n-m` and calling procedure `match` in the for-loop `iterate` in lines 19 to 21. As a shorthand, we write `m` and `n` for the size of the pattern and text, respectively.

Procedure `match` begins with calling procedure `compare` to match `P` to `T` beginning at shift `s` with the initial index `i = 0`. If `compare` returns with `i = m` (end of `P` is reached), the match succeeds, and `s` is a valid shift; otherwise, the match fails (end of `P` is not reached). The then-branch pushes the valid shift `s` to `R` and resets `i` to zero. Line 11 is required to restore the last value of `s` from the top of `R` because the last `push` *moved* that value to `R` and thereby zero-cleared `s`. (This point is explained below.) In the else-branch, after the match fails at `i < m`, the computation of `compare` is undone by uncalling `compare` to reset `i` to its initial value `0`. (This is also explained below.) A local scope for `i` is opened and closed at the beginning and end of `match` using a `local–delocal` declaration. Furthermore, this scope declaration asserts the initial and final values of local variable `i` (in both cases `i = 0`).

Procedure `compare` compares `P` with `T` at `s`, beginning with index `i = 0`. The loop begins with an entry test at line 2, which asserts that initially `i = 0`, and ends at line 4 when `T[s+i]` $\neq$ `P[i]`. This loop always terminates because by convention the terminating value `P[m]` is not in `T`.

Reversible Programming Similar to the other language paradigms, reversible computing has its own programming methodology. We summarize the programming techniques relevant to the programs in this study and exemplify them with examples from the programs. This is related to three important reversible programming themes: control flow, reversible updates, and data structures.

```
1  procedure compare(int T[], P[], s, i)
2    from i = 0 loop          // index assertion
3      i += 1                 // character-by-character comparison
4    until T[s+i] != P[i]     // loop terminates when a mismatch occurs
5
6  procedure match(int T[], P[], s, stack R)
7    local int i = 0
8    call compare(T, P, s, i)
9    if i = m then  // match succeeded
10     push(s, R)   // push s to stack R of valid shifts w/ clearing s
11     s += top(R)  // restore the value of s
12     i -= m       // clear i
13   else           // match failed
14     uncall compare(T, P, s, i) // clear i
15   fi s = top(R)  // current shift s is valid
16   delocal int i = 0
17
18 procedure naivesearch(int T[], P[], stack R)
19   iterate int s = 0 to n-m  // slide over text
20     call match(T, P, s, R)  // match at current shift s
21   end
```

Figure 1: Reversible naive string-matching algorithm.

Control flow: Join points in the control flow of a program require assertions to make them *backward deterministic*. In reversible languages, each join point is associated with a predicate that provides an assertion regarding the incoming computational states. This suggests that we must identify a predicate that is true when coming from a then-branch and false when coming from an else-branch. For a loop, we must identify a predicate that is initially true and false after each iteration. These assertions regarding the incoming control flow ('*come from*') are evaluated at runtime, similar to the tests that dispatch the outgoing control flow ('*go to*'). If a predicate does not have the expected truth value, the control-flow operator is undefined, and therefore the entire program.

Examples are `fi`-predicates in *reversible conditional* (`if-fi`) and `from`-predicates in a *reversible while-loop* (`from-until`). Sometimes, these assertions are easy to find, such as the entry test in line 2 of the increment loop in Fig. 1, which checks that the loop begins from `i` $= 0$ and `i` $\neq 0$ after the first iteration. The exit test `s = top(R)` in line 15 of the conditional uses the fact that the shifts in stack `R` are unique. Thus, whenever a match fails, indicating that `s` is not pushed to `R`, the current shift `s` and the last shift `top(R)` differ. An excerpt from the program highlights these two cases:

```
from i = 0 loop          if i = m then push(s,R) ...
  i += 1                   else ... no push ...
until T[s+i] != P[i]     fi s = top(R)
```

However, these assertions are not always easy to find and may require a restructuring of the program. Only a few conventional control-flow operators are reversible and do not require additional assertions, such as for-loops that iterate for a fixed number of times, *e.g.*, `iterate` in lines 19–21.

Reversible updates: Data can only be reversibly updated. The usual computational resources for deleting data in one way or another are not available (*e.g.*, forgetting local variables upon procedure re-

turn). We present several update techniques used in our programs starting from a straightforward initialization of a zero-cleared variable to the uncalling of a procedure to reset values. Readers interested in reversible updates defined in a more general form should refer to [21].

(i) Copying & zero-clearing. If variable `i` is known to be zero, it can be set to a value, *e.g.*, by addition. For example, `i += m` has the effect of reversibly copying the value of `m` to `i`. Similarly, if we know that `i` has the same value as `m`, that is `i` = `m`, we can zero-clear `i` using `i -= m`. However, the relationship between two variable values is not always known. Additionally, when it becomes known owing to an equality test in a conditional, we can exploit this knowledge in the then-branch to zero-clear the variable. This is used in line 12 to reversibly reset `i` to zero:

```
if i = m then ...
  i -= m
else ...
fi ...
```

These techniques are indirectly used in a local–delocal declaration, where the local variable is initialized and cleared at the beginning and end of its scope using an equality test (here, however, just a simple `x = 0` in lines 7 and 16). In general, the declaration of a local variable, `i`, has the following form, where `i` is initially set to the value of e and in the end must have a value equal to the value of e':

```
local int i = e ... delocal int i = e'
```

(ii) Compute-uncompute. Reversible programs are forward and backward deterministic; thus, they can run efficiently in both directions. Many reversible languages not only provide access to their standard semantics, *e.g.*, using a procedure call, but also to their inverse (backward) semantics, *e.g.*, using a procedure uncall. An uncall of a procedure is as efficient as a call because a procedure is forward and backward deterministic. We employ this property to reset index `i` after an unsuccessful match, which can occur at any position `i` < `m` in a pattern. We cannot determine the subtrahend to zero-clear `i` (and we cannot use the irreversible `i -= i`). Instead we undo the computation of `i` by an uncall in the else-branch. This resets `i` to its initial value `0`. By combining the techniques seen so far, we ensure that `i` is zero-cleared after the `if-fi`. In the then-branch, we use the equality `i = m`; in the else-branch we undo the computation:

```
local int i = 0
  call compare(...,i)
  if i = m  then ...
    i -= m
  else uncall compare(...,i)
  fi ...
delocal int i = 0
```

The compute-uncompute method goes back to the first RTMs [3], where a machine is textually composed with its inverse machine to restore the original computation state (which doubles the size of the entire machine). The call–uncall method above shares the text of a procedure (here, `compare`). It just invokes the standard resp. inverse computation of the procedure. We could have used an uncall in the then-branch. Instead, we exploit the knowledge about `i = m` from the entry test of the `if-fi` to zero-clear `i` using `i -= m`, which takes constant time, whereas the uncall requires time proportional to the length of `P`. The conditional takes advantage of both techniques.

Bennett used *program inversion* to obtain an inverse RTM, whereas the aforementioned method uses *inverse computation*. We could have used program inversion to invert the procedure `compare` into the inverse procedure `compare`$^{-1}$ and invoked the latter using `call compare`$^{-1}$ to reset `i`. Both methods, calling the inverse procedure `compare`$^{-1}$ and inverse computation of `compare`,

```
call compare^-1(...,i)  and  uncall compare(...,i),
```

are functionally equivalent. Because RTMs cannot access their inverse semantics, *e.g.*, by an uncall, program inversion is the only choice to build RTMs that restore the input from their output, whereas in a reversible language typically both choices are available. We refer to them collectively as the *compute–uncompute* programming method. It is used in many forms at all levels of a reversible computing system from reversible circuits (*e.g.*, [20]) to high-level languages (*e.g.*, [14]).

Data structures: The data structures in reversible languages are the same as in conventional languages, such as arrays, stacks and lists, only the update operations on the data structures must be reversible. In the case of a stack, the operations push and pop can be defined as inverse to each other by letting them swap in and out the value on top of the stack, which means that $pop = push^{-1}$ [23]:

$$(v, v_n \dots v_1) \underset{pop}{\overset{push}{\rightleftarrows}} (0, v\ v_n \dots v_1) \tag{3}$$

This definition of a push has the unfamiliar property that `push(s,R)` in line 10 moves the value v of `s` to the top of the stack `R` and zero-clears `s`. Because we need the value that we have pushed to continue the search, the value is copied back to `s` from the top of `R` using `s += top(R)` in line 11.

We can now complete the body of procedure `match` in lines 7–16 by adding the two statements to the then-branch and the exit test that we discussed above to `fi`:

```
local int i = 0
  call compare(...,i)
  if i = m then
    push(s,R)
    s += top(R)
    i -= m
  else uncall compare(...,i)
  fi s = top(R)
delocal int i = 0
```

We remark that abstract data types and object-oriented features can be used in reversible languages provided that their update operations and methods are reversible. Ideally, they are designed such that call and uncall can be used. When operators are inverse to each other, only one of them needs to be implemented. The idea of *code sharing* by running code backward can be traced back to the 60s [18].

We have presented all reversible-programming techniques used in the procedures `naivesearch`, `match`, and `compare`. This completes the review of the reversible naive string-matching algorithm shown in Fig. 1.

3 A Reversible Rabin–Karp Algorithm

The Rabin–Karp algorithm [10] replaces exact matches with approximate matches, which are inexact but fast, and performs exact matches only if a successful match is possible. This makes the algorithm conceptually easy and fast in practice. This study refers to the version presented by Cormen et al. [4].

The algorithm extends the naive string-matching algorithm by performing at each shift s, an approximate match by comparing the *hash values* of P and T at s. An exact match (by procedure `match`) can only succeed if the two hash values are identical; otherwise, an exact match is impossible. In either case, the next hash value at $s+1$ can be computed in constant time from the current hash value at s (rolling hash) using a *hash-update function* ϕ_s. Hash values typically fit into single words that can be compared in constant time. The initial hash values of P and T at shift 0 are computed at the beginning of the algorithm using a *hash function* (pre-processing). The subsequent hash values are then computed using the hash-update function.

The key to an efficient clean reversible Rabin–Karp matcher is a reversible constant-time calculation of the rolling hash values. We have explained the reversible naive string-matching algorithm in the previous section, and we can reuse procedures `match` and `compare` from Fig. 1 for the reversible Rabin–Karp algorithm. In this section, we focus on the injectivization and implementation of the hash functions that show the considerations for the development of a more advanced reversible algorithm.

A preliminary version of the reversible Rabin–Karp program has appeared as a technical report [19]. The reversible Rabin–Karp program in this study becomes more concise and modular because of the use of macros and iterate loops.

Hash Function The Rabin–Karp algorithm requires a pre-process that calculates the hash values of the given pattern, $P[0..m-1]$, and of the initial substring, $T[0..m-1]$, of the given text T. The initial substring has the same length m as P. Recall that P and T are two integer arrays over the non-empty alphabet of d integers $\{0,\ldots,d-1\}$.

Let p denote the hash value of P obtained using a polynomial hash function with modulus q:

$$p = (P[0]d^{m-1} + P[1]d^{m-2} + \cdots + P[m-1]) \bmod q\,. \tag{4}$$

Similarly, let t_s denote the hash value of the substring $T[s..s+m-1]$ of length m at shift s:

$$t_s = (T[s]d^{m-1} + T[s+1]d^{m-2} + \cdots + T[s+m-1]) \bmod q\,. \tag{5}$$

The polynomials can be computed in $\Theta(m)$ using Horner's rule. Preferably, modulus q should be as large a prime as possible such that dq fits into a single word: thus, all modulo operations are single-precision arithmetic.

The following properties of the two hash values are important. If $t_s \neq p$, $T[s..s+m-1] \neq P[0..m-1]$: thus, shift s cannot be valid. If $t_s = p$, it is *possible* that $T[s..s+m-1] = P[0..m-1]$: thus, an exact match is required to determine if shift s is valid.

Whereas hash value p of P is the same during the matching, hash value t_s of T must be calculated for each shift s. In order to efficiently calculate hash values t_s for $s > 0$, we calculate the hash value t_{s+1} at the subsequent shift $s+1$ from the hash value t_s at the current shift s, using a recurrence function. We use this function to reversibly update the hash values in constant time. For a reversible implementation, conditions are first determined under which the function is injective. Then the function is rewritten into a composition of modular arithmetic operators, each of which is embedded in a reversible update.

Hash-Update Function We can compute t_{s+1} from t_s for $0 \leq s \leq n-m$ because

$$t_{s+1} \quad = \quad \phi_s(t_s) \tag{6}$$

where the recurrence function ϕ_s is defined as

$$\phi_s(x) \quad = \quad (d(x - T[s]d^{m-1}) + T[s+m]) \bmod q\,. \tag{7}$$

The recurrence function calculates the subsequent hash value from the current hash value x by canceling the old highest-order radix-d digit $T[s]$ via subtraction, shifting the value via multiplication with d, adding the new lowest-order radix-d digit $T[s+m]$, and obtaining its remainder when divided by q. We can compute $\phi_s(t_s)$ in constant time if factor d^{m-1} is precomputed. For reversible computing it is problematic that ϕ_s is generally *not injective* because the modulo operation is not injective in its first argument for arbitrary integers. Determining the conditions under which a function becomes injective by exploiting its properties and specific application context is an important step in the development of a clean reversible algorithm.

We exploit certain properties in the domain of arithmetic operations in ϕ_s. The congruence of two integers x and y modulo q, $x \equiv y \pmod q$, is compatible with addition, subtraction, and multiplication. But unlike these operations, division cannot always be performed. To show the injectiveness of ϕ_s in the following lemma requires that d and q be *coprime* integers, which means their greatest common divisor is 1. For example, when d and q are not coprime, *e.g.*, $d = 2$ and $q = 6$ then $2 \cdot 1 \equiv 2 \cdot 4 \pmod 6$, we cannot divide this congruence by 2 because $1 \not\equiv 4 \pmod 6$. Whereas if they are coprime, *e.g.*, $d = 2$ and $q = 3$, then $2 \cdot 1 \equiv 2 \cdot 4 \pmod 3$ and we can divide this by 2 to deduce $1 \equiv 4 \pmod 3$.

We employ this constraint on the operands in Lemma 3, which establishes that ϕ_s is injective. For the cancelation of common terms in congruences modulo q, we use the following two lemmas in the proof.

Lemma 1 (*e.g.* [13, Prop. 13.3])**.** *Suppose $x, y \in \mathbb{Z}$ and $x \equiv y \pmod q$. If $c \in \mathbb{Z}$ then*

$$x + c \equiv y + c \pmod q \implies x \equiv y \pmod q\,. \tag{8}$$

Lemma 2 (*e.g.* [13, Prop. 13.5])**.** *Suppose d and q are coprime integers. If $x, y \in \mathbb{Z}$ then*

$$dx \equiv dy \pmod q \implies x \equiv y \pmod q\,. \tag{9}$$

Proof. We assume that $dx \equiv dy \pmod q$. Therefore, $d(x-y)$ is a multiple of q. Because d and q are coprime, $(x-y)$ is a multiple of q, *i.e.* $x \equiv y \pmod q$. □

Lemma 3. *Provided that $0 \leq x < q$, $0 < d < q$, and d and q are coprime, recurrence function ϕ_s is injective, where*

$$\phi_s(x) \quad = \quad (d(x - T[s]d^{m-1}) + T[s+m]) \bmod q\,. \tag{10}$$

Proof. We show that for any x_1 and x_2, whenever $\phi_s(x_1) = \phi_s(x_2)$, we have $x_1 = x_2$.

$$\begin{aligned}
& \phi_s(x_1) \;=\; \phi_s(x_2) \\
\implies\; & d(x_1 - T[s]d^{m-1}) + T[s+m] \;\equiv\; d(x_2 - T[s]d^{m-1}) + T[s+m] \pmod q \\
\implies\; & d(x_1 - T[s]d^{m-1}) \;\equiv\; d(x_2 - T[s]d^{m-1}) \pmod q \qquad \because \text{Lemma 1} \\
\implies\; & x_1 - T[s]d^{m-1} \;\equiv\; x_2 - T[s]d^{m-1} \pmod q \qquad \because \text{Lemma 2 and } d \text{ and } q \text{ are coprime integers} \\
\implies\; & x_1 \;\equiv\; x_2 \pmod q \qquad \because \text{Lemma 1} \\
\implies\; & x_1 \;=\; x_2 \qquad \because 0 \leq x_1, x_2 < q
\end{aligned}$$

□

The condition that d and q are coprime is not a restriction in practice because q is usually selected to be a large prime number. Thus, if q is prime, any alphabet size $0 < d < q$ can be used.

Injective Modular Arithmetic For efficient calculation of a hash value, it is preferable to perform modular arithmetic at each elementary arithmetic operation instead of first calculating a large value that may exceed the largest representable integer. Therefore, we define the injective recurrence function ϕ_s using elementary modular arithmetic operators, each of which is sufficiently simple to be easily implemented in a reversible language (*e.g.*, supported as built-in operators or by reversible hardware). We rewrite $\phi_s(x)$ in Eq. (10) into a composition of *modular arithmetic operators* $+_q$, $-_q$, and $\cdot_q$ using the distributivity of the modulo operation as shown in Eq. (11). Similarly, the hash functions in Eqs. (4) and (5) can be rewritten as Eqs. (13) and (14). The same transformation is required for implementation in conventional languages. Additionally, we inspect the injectivity of each operator such that a reversible implementation can be provided.

The injective function, ϕ_s, comprises the following operators:

$$\phi_s(x) \quad = \quad d \cdot_q (x -_q T[s] \cdot_q h) +_q T[s+m] \tag{11}$$

where factor $h = d^{m-1} \pmod q$ is precomputed by

$$h \quad = \quad \underbrace{d \cdot_q \cdots \cdot_q d}_{m-1}. \tag{12}$$

Hash value p of a given pattern P and initial hash value t_0 of a given text T can be obtained using Horner's rule defined in ψ:

$$p \quad = \quad \psi(P[0..m-1]) \tag{13}$$

$$t_0 \quad = \quad \psi(T[0..m-1]) \tag{14}$$

where

$$\psi(X[i..j]) \quad = \quad X[j] +_q d \cdot_q (X[j-1] +_q d \cdot_q (\cdots +_q d \cdot_q (X[i+1] +_q d \cdot_q X[i]) \cdots)). \tag{15}$$

The identity between ψ applied to the final substring, $T[n-m..n-1]$, at shift $n-m$ and the hash update ϕ_{n-m-1} is used to zero-clear the final hash value, t_{n-m}, in the reversible program:

$$t_{n-m} \quad = \quad \psi(T[n-m..n-1]) \qquad \because \text{Eq. (15)} \tag{16}$$

$$\quad = \quad \phi_{n-m-1}(t_{n-m-1}) \qquad \because \text{Eq. (6)}. \tag{17}$$

The problem is that modular arithmetic operations are generally non-injective. However, under certain conditions, they are injective in one of their arguments. For example, they are injective in their first arguments:[2]

- Addition $x +_q y$ and subtraction $x -_q y$ are injective in their first arguments if $0 \leq x, y < q$.
- Multiplication $x \cdot_q d$ is injective in its first argument if $0 \leq x < q$, $1 \leq d < q$, and d and q are coprime.

Note that $x \cdot_q d$ is not injective in its first argument, unless d and q are coprime. Thus, the relationship between d and q is a necessary condition. Analogously, the same holds for the second arguments of $+_q$, $-_q$, and $\cdot_q$. Recall that the composition of injective functions is an injective function. Thus, the injectiveness of operators $x +_q y$, $x -_q y$, and $x \cdot_q y$ under the stated conditions demonstrates the injectiveness of Eq. (11) under corresponding conditions.

[2] A partial function, $f : X \times Y \times \cdots \times Y \rightharpoonup Z$, is *injective in its first argument* iff $\forall x_1, x_2 \in X, \forall y_1, \ldots, y_n \in Y$: if $f(x_1, y_1, \ldots, y_n)$ and $f(x_2, y_1, \ldots, y_n)$ are defined, $f(x_1, y_1, \ldots, y_n) = f(x_2, y_1, \ldots, y_n) \implies x_1 = x_2$. Similarly, for other arguments [21].

Implementation of Modular Arithmetic A ternary function that is injective in its first argument $f(x,y,z)$ can be embedded in the *reversible update* $g(x,y,z) = (f(x,y,z),y,z)$. Arguments y and z are part of the result of g. Thus, g is injective. Such reversible updates for $x +_q y$ and $x \cdot_q y$ can be implemented in Janus. We write

- `x +=q y` for $g_1(x,y,q) = (x +_q y, y, q)$, and
- `x *=q y` for $g_2(x,y,q) = (x \cdot_q y, y, q)$.

In the implementation of these operators in a reversible language, `x`, `y`, and `q` are integers that cannot be larger than the largest representable integer in that language. Otherwise, the same restrictions as those for mathematical modular arithmetic apply. Thus, we assume that `x` and `y` range over 0 to `q`-1, except that `y` ranges from 1 in multiplication. It is the programmer's responsibility to ensure `y` and `q` are coprime integers in `x *=q y`. In practice, it is sufficient that `q` is prime, so that `y` and `q` are coprime. The subtraction, $x -_q y$, can be realized using an uncall to `x +=q y`, and is written as `x -=q y`. The variable on the left-hand side must not occur on the right-hand side of any of these operators. We assume that these operators perform in constant time.

The `n`th power of `b` can be stored in a zero-cleared variable, `z`, written `z +=q b^n`, by initializing `z` with `1` and repeating `n`-times `z *=q b`:

```
z += 1
iterate int i = 0 to n-1
  z *=q b
end
```

For notational simplicity, we write `z -=q b^n` as the inverse of `z +=q b^n` to zero-clear `z`. In an implementation the arguments of modular arithmetic operators cannot be larger than the largest representable integer in a particular reversible programming language.

A Reversible Rabin–Karp Matcher Figure 2 shows the program for the reversible Rabin–Karp algorithm. The program consists of three procedures and uses procedure `match` in Fig. 1.

The main procedure `rabinkarp` is called with text `T`, pattern `P`, alphabet size `d` (including the dummy character terminating `P`), modulus `q`, and an initially empty stack `R` as the input. When it returns, all valid shifts are stored in `R`, and all other variables `T`, `P`, `d`, and `q` remain unchanged. Therefore, it is an implementation of an input-preserving injectivization of the string-matching problem shown in Eq. 2.

The main iteration in lines 18–23 corresponds to that in the main procedure of the naive string-matching algorithm, except that the hash value `p` of `P` and the hash value `t` of `T` at shift `s` are compared. Only if a match is possible, that is `p = t` is true in line 19, an exact match of `P[0..m-1]` and `T[s..s+m-1]` is performed in the then-branch by calling `match` in line 20. This exact match can update `R` with a valid shift depending on the outcome of the comparison. After the conditional, `t` at shift `s` is updated to the hash value at shift `s+1` through a call to `update`. Subsequently, the iteration continues at the next shift.

The pre- and post-processing before and after the main iteration are performed in lines 14–16 and lines 25–27, respectively. In pre-processing, the hash values `p` and `t` are initialized by the calls to `init_h` as defined in Eqs. (13, 14), and `h` is precomputed using `h +=q d^m-1` as defined in Eq. (12). Post-processing, a typical idiom of reversible programming to zero-clear variables, uncomputes the values of `h`, `t` and `p`. Notably, the call of `init_h` in line 15 and the uncall of `init_h` in line 26 have different indices. The uncall uses the fact that the last value of `t` is the hash value of `T` in the *last shift* `n-m` (*cf.*, Eq. (17)), whereas the initial value of `t` is the hash value of `T` in the *first shift* `0`.

```
1  procedure init_h(int x, i, j, X[], d, q)
2    iterate int k = i to j-1 // compute hash value by Horner's rule
3      x *=_q d      // shift by one radix-d digit
4      x +=_q X[k]   // add the low-order radix-d digit
5    end
6
7  procedure update(int t, T[], s, h, m, q)
8    t -=_q T[s]*h   // remove the high-order radix-d digit
9    t *=_q d        // shift by one radix-d digit
10   t +=_q T[s+m]   // add the low-order radix-d digit
11
12 procedure rabinkarp(int T[], P[], d, q, stack R)
13   local int t=0, int p=0, int h=0
14   call init_h(p, 0, m, P, d, q) // store hash of P[0..m-1] in p
15   call init_h(t, 0, m, T, d, q) // store hash of T[0..m-1] in t
16   h +=_q d^(m-1)                // precompute factor h
17
18   iterate int s = 0 to n-m      // slide over text
19     if p = t then               // compare hash values
20       call match(T, P, s, R)    // match at current shift s
21     fi p = t
22     call update(t, T, s, h, m, q)    // update hash value
23   end
24
25   h -=_q d^(m-1)                      // clear h
26   uncall init_h(t, n-m, n, T, d, q)   // clear t
27   uncall init_h(p, 0,   m, P, d, q)   // clear p
28   delocal int t=0, int p=0, int h=0
```

Figure 2: Reversible Rabin–Karp algorithm.

Procedure `update` computes the subsequent hash value, t_{s+1} from t_s, in constant time using Eq. (11). Line 8 removes the high-order radix-`d` digit from `t`, line 9 multiplies `d`, which shifts the radix-`d` number left by a one-digit position, and line 10 adds the low-order radix-`d` digit. All the operations are modulo `q`.

Procedure `init_h` computes in an initially zero-cleared variable `x`, the hash value $\psi(X[i..j-1])$ of a substring $X[i..j-1]$, using Horner's rule as shown in Eq. (15). In addition to `x`, the indices `i` and `j`, array `X`, alphabet size `d`, and modulus `q` are used as inputs. The hash values $p = \psi(P[0..m-1])$ and $t_0 = \psi(T[0..m-1])$ are computed in $\Theta(m)$.

Space, Time, and Reversibilization Regarding the space consumption of the program, no extra arguments are passed to procedure `rabinkarp` to maintain garbage values. All local variables are allocated and deallocated in the body of the procedures, and neither stacks nor arrays are allocated. Therefore, no additional space is required compared with the irreversible original of the program [4].

The pre- and post-processing times are bounded by the running time of `init_h`, which is $\Theta(m)$. The worst-case running time of matching is $\Theta((n-m+1)m)$, which is the same as that of the irreversible Rabin–Karp algorithm. In the worst case, the iteration repeats the $\Theta(m)$ steps of the exact match $n-m+1$ times. The shift `s` increases monotonically in the main iteration of procedure `rabinkarp` and a limited number of elements $T[s..s+m-1]$ of the text is accessed at each iteration. Thus, just like the irreversible

original, the proposed Rabin–Karp algorithm computes over bounded space.

Notably, no extra space is required by the two reversible string-matching programs. The speedup gained by allowing deletion as a computational resource (at the expense of additional heat dissipation owing to entropy increase [6, 11]) is a constant factor of approximately two (the uncall in procedure `match` in case of an unsuccessful comparison). Thus, if reversibility were removed and the two programs were turned back into C-like imperative programs, speed, but no space would be obtained. The reversible programs, developed in this study, are in contrast to what one obtains from mechanically reversibilizing the string-matching algorithms using Bennett's method [3]. Bennett's method has the advantage that it can be applied to any irreversible program; however, it requires additional space proportional to the length of the computation because of the recording of a trace (for reversible simulations with improved space efficiency, see *e.g.*, [17]). Therefore, the entire run of a reversibilized string matcher, including all hash calculations and mismatches, is recorded in the computation history. Moreover, because of the uncompute phase added by Bennett's method to clean up the trace after finding all valid shifts, the reversible string-matching program produced by this reversibilization method is no longer computing over bound space.

4 Conclusion

In this study, we have designed and implemented the reversible versions of a naive string-matching algorithm and the Rabin–Karp algorithm. We have shown that the hash-update function with a reasonable restriction in the reversible Rabin–Karp algorithm is injective. The reversible versions of a naive string matching algorithm and the Rabin–Karp algorithm have the same asymptotic running time $\mathrm{O}((n-m+1)m)$ and space usage $\mathrm{O}(n+m)$, as the irreversible versions. Because the original inputs preserved over the runs are not regarded as garbage, both reversible algorithms are clean, *i.e.* they produce no garbage as output. The main iteration monotonically increases the shift `s` from 0 to `n-m+1`. Thus, the proposed Rabin–Karp algorithm is a streaming algorithm. This property cannot be automatically obtained by the reversibilization of Bennett [3] and Lange–McKenzie–Tapp [12]. It is expected that the reversible algorithms developed in this study can be a part of other algorithms, and the insights gained from constructing the reversible algorithms can be applied in future program developments. Verifying conventional programs is not always an easy task (*e.g.*, [5]), and exploring the reversibilization and mechanical verification of reversible programs will be a challenge in future work.

Acknowledgments The authors would like to thank Geoff Hamilton, Temesghen Kahsai, and Maurizio Proietti for their kind invitation to contribute to the workshop HCVS/VPT at ETAPS week 2022 in Munich and the anonymous reviewers for the useful feedback. The idea for the reversible algorithms in this study was brewed in joint work with Kaira Tanizaki and Masaki Hiraku. The second author was supported by JSPS KAKENHI Grant Number 22K11983.

References

[1] Holger Bock Axelsen & Robert Glück (2011): *A simple and efficient universal reversible Turing machine*. In Adrian-Horia Dediu, Shunsuke Inenaga & Carlos Martín-Vide, editors: *Language and Automata Theory and Applications. Proceedings*, *LNCS* 6638, Springer-Verlag, pp. 117–128, doi:10.1007/978-3-642-21254-3_8.

[2] Holger Bock Axelsen & Tetsuo Yokoyama (2015): *Programming techniques for reversible comparison sorts*. In Xinyu Feng & Sungwoo Park, editors: *APLAS*, *LNCS* 9458, Springer-Verlag, pp. 407–426, doi:10.1007/978-3-319-26529-2_22.

[3] Charles H. Bennett (1973): *Logical reversibility of computation*. *IBM J. Res. Dev.* 17(6), pp. 525–532, doi:10.1147/rd.176.0525.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein (2009): *Introduction to Algorithms*, 3rd edition. MIT Press, Cambridge, MA.

[5] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2022): *Contract strengthening through constrained Horn clause verification*. In: *HCVS/VPT'22*, pp. 23–34. EPTCS (to appear).

[6] Alexis De Vos (2020): *Endoreversible models for the thermodynamics of computing*. *Entropy* 22(6), p. Article 660, doi:10.3390/e22060660.

[7] Robert Glück & Andrei V. Klimov (1994): *Metacomputation as a tool for formal linguistic modeling*. In Robert Trappl, editor: *Cybernetics and Systems '94*, 2, World Scientific, pp. 1563–1570.

[8] Robert Glück & Tetsuo Yokoyama (2019): *Constructing a binary tree from its traversals by reversible recursion and iteration*. *IPL* 147, pp. 32–37, doi:10.1016/j.ipl.2019.03.002.

[9] Robert Glück & Tetsuo Yokoyama (2022): *Reversible computing from a programming language perspective*. *Theor. Comput. Sci.* In Press.

[10] Richard M. Karp & Michael O. Rabin (1987): *Efficient randomized pattern-matching algorithms*. *IBM J. Res. Dev.* 31(2), pp. 249–260, doi:10.1147/rd.312.0249.

[11] Marina Krakovsky (2021): *Taking the heat*. *Commun. ACM* 64(6), pp. 18–20, doi:10.1145/3460214.

[12] Klaus-Jörn Lange, Pierre McKenzie & Alain Tapp (2000): *Reversible space equals deterministic space*. *J. Comput. Syst. Sci.* 60(2), pp. 354–367, doi:10.1006/jcss.1999.1672.

[13] Martin Liebeck (2015): *A Concise Introduction to Pure Mathematics*, 4th edition. Chapman and Hall/CRC.

[14] Torben Æ. Mogensen (2022): *Hermes: A reversible language for lightweight encryption*. *Science of Computer Programming* 215, p. Article 102746, doi:10.1016/j.scico.2021.102746.

[15] Keisuke Nakano (2022): *Time-symmetric Turing machines for computable involutions*. *Science of Computer Programming* 215, p. Article 102748, doi:10.1016/j.scico.2021.102748.

[16] Francisco Orts, Gloria Ortega, Elías F. Combarro & Ester M. Garzón (2020): *A review on reversible quantum adders*. *J. Netw. Comput. Appl.* 170, p. 102810, doi:10.1016/j.jnca.2020.102810.

[17] Tommi Pesu & Iain Phillips (2015): *Real-time methods in reversible computation*. In Jean Krivine & Jean-Bernard Stefani, editors: *RC*, *LNCS* 9138, Springer, pp. 45–59, doi:10.1007/978-3-319-20860-2_3.

[18] Edwin D. Reilly, Jr. & Francis D. Federighi (1965): *On reversible subroutines and computers that run backwards*. *Commun. ACM* 8(9), pp. 557–558, 578, doi:10.1145/365559.365593.

[19] Kaira Tanizaki, Masaki Hiraku & Tetsuo Yokoyama (2022): *Reversibilization of the naive string-match algorithm and the Rabin–Karp algorithm*. *Academia. Sciences and Engineering: Journal of the Nanzan Academic Society* 22(3), pp. 124–132, doi:10.15119/00003946. In Japanese.

[20] Michael Kirkedal Thomsen, Holger Bock Axelsen & Robert Glück (2012): *A reversible processor architecture and its reversible logic design*. In Alexis De Vos & Robert Wille, editors: *RC*, *LNCS* 7165, Springer-Verlag, pp. 30–42, doi:10.1007/978-3-642-29517-1_3.

[21] Tetsuo Yokoyama, Holger Bock Axelsen & Robert Glück (2008): *Principles of a reversible programming language*. In: *Computing Frontiers. Proceedings*, ACM Press, pp. 43–54, doi:10.1145/1366230.1366239.

[22] Tetsuo Yokoyama, Holger Bock Axelsen & Robert Glück (2016): *Fundamentals of reversible flowchart languages*. *Theor. Comput. Sci.* 611, pp. 87–115, doi:10.1016/j.tcs.2015.07.046.

[23] Tetsuo Yokoyama & Robert Glück (2007): *A reversible programming language and its invertible self-interpreter*. In: *PEPM*, ACM Press, pp. 144–153, doi:10.1145/1244381.1244404.